DESIGN AND IMPLEMENTATION OF A WORKFLOW MODEL

Part II : Object Model

By

M. KALADHAR

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY, 1997

Design and Implementation of a workflow model

Part II : Object Model

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

M. Kaladhar

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

Jan. 1997

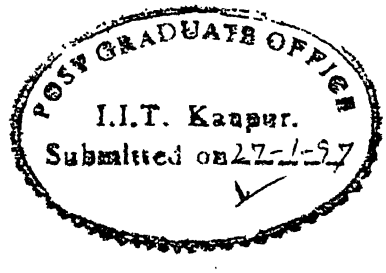
CSE-1997-M-KAL-DES

CERTIFICATE

This is to certify that the work contained in the thesis titled *Design and Implementation of a Workflow, Part II : Object Model* by **M. Kaladhar** has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. Tapas K. Nayak,
Assistant Professor,
Dept. of CSE,
IIT Kanpur.

Dr. Ratan K. Ghosh,
Associate Professor,
Dept. of CSE,
IIT Kanpur.



Acknowledgements

I would like to express my sincere gratitude to Dr. Tapas Nayak and Dr. R K Ghosh. I extend my thanks to Dr. Tapas Nayak for his valuable guidance and encouragement during the thesis work.

Thanks are due to Mr. Srinivas for his constant help without whom i could not have completed the work in time.

My special thanks to Mr. G G S Achary for his invaluable encouragement through out my stay in IIT Kanpur.

Many of my friends giri, gsri, shankar, chitti, naidu, dsri, lade, raghu and all helped me in numerous ways in this endeavour and their assistance is greatfully appreciated.

I am also greatful for the ideas obtained from M Tech 95 CS batch.

I cannot express express the patience and understanding of my parents during the last two years while i was away from them.

Finally . . . . . thanx to csealpha3 !!

Abstract

A Workflow management system (WFMS) automates the flow of control and data between tasks, and ensures the tasks are executed only by the valid users. Existing advanced transaction models solve only part of the problem. The most important feature of WFMS is their ability to describe an organization and adapt the definition and execution of workflow processes to the particular characteristics of that organization.

This thesis and its companion [13] proposes a workflow model which allows passing of objects between tasks. The control flow between the tasks are defined in a task specification language, and the work objects in the system are defined using object definition language. We have developed a prototype of the workflow model with **Ingres** as the back end. GUI tools have been developed for specifying and tracking the objects in the workflow.

The work has been divided into two parts : the task model and object model. The task model is presented in the companion thesis [13]. The object model and workflow tools are covered in this thesis. The object model includes the specification of the object language and maintaining the queues of objects for various tasks. It involves the interaction with the RDBMS for storage and retrieval of objects. The specification tool is used for defining the workflow. The monitoring tool used for querying (also monitoring) the tasks and work objects in the workflow.

Contents

1 Introduction 1

1.1 What is **workflow?** : : :

4.2.1 The stack machine : 25

5 The Object Server 28
5.1 SchemaBase : 30
5.2 ObjectBase : 32
5.3 Object Server API : 34
5.4 Queue Management : 35
5.4.1 Structure of the Object ID : 36
5.4.2 Lock Management : 36
5.4.3 Table Structure of Qserver : 37
5.4.4 Implementation of Server API : : : : : : : : : : : : : : : : : : : 37

6 Workflow Tools 42
6.1 Workflow specification tool : 42
6.2 Workflow monitoring tool : 46
6.2.1 Task monitoring : 47
6.2.2 Work object monitoring : 47

7 Conclusions 49
7.1 Future Work : 49

A Object Language Grammer 51

Bibliography 58

CHAPTER 1

Introduction

Modern business applications in current data processing environment involve operations and interactions on multiple systems. Examples of such multi-system applications include travel reservation, service order processing, and transactions in a bank. Transaction-oriented systems provide only limited support for managing the control and data flow in multi-system applications. The task coordination structure in transaction systems are difficult to understand and manage, and it is scattered through the application program code. A workflow model uses a declarative control and data flow specification language to facilitate a separation between the application code, and the control and data flows between different tasks of the application.

1.1 What is workflow?

Workflows are activities involving coordinated execution of multiple *tasks* performed by different *processing entities* [10]. A *task* defines some work to be done and can be specified in a number of ways, including an e-mail, a form, a message, or a program. A processing entity that performs the tasks may be a person or software system. The *Workflow management system*(WFMS) primarily concerned with the specification and scheduling of various tasks in the workflow.

A task is a minimal unit of work (computation) that defines a semantically meaningful portion of overall work, and therefore is indivisible from the point of view of

the workflow application. Each task performs some operations on its underlying (database) system. Therefore, a task is typically an application program (transaction) to do some well-defined processing of objects. A task may be interactive or non-interactive one. For example, the signature verification task requires interaction with the user, whereas the account updating task is non-interactive.

In a typical work environment, the tasks may be **distributed** across a network of machines. A task can be specified using a role, this means all the persons that fit in that role are eligible to execute the task.

Regular users interact with the system using *worklists*. A worklist is a list of jobs (work items) associated with the user. A job is a unit of work performed by a task e.g, a cheque in a bank application. When a user selects a job for execution, it disappears from all other worklists. Each task performs some operations on the object and is passed to the next task in the pipeline. The routing of objects between the tasks depends on the application. Some of the tasks may generate copies of objects and route them independently. These copies are later merged at some other task. This is how essentially work objects *flow* through the system.

1.2 History and Related Work

The idea of a workflows can be traced to *Job Control Language* (JCL) of batch operating system, such as OS, which allowed the user to specify a job as a collection of steps. Each step was an invocation of a program, and some steps could be executed conditionally - for example, only if previous steps was successful or if it is failed. This simple idea was subsequently expanded in many products and research prototypes. The extensions allow the designer to specify the data and control flow among the tasks and to selectively choose transactional characteristics of the activity, based on its semantics.

Groupware such as Lotus Notes provide sophisticated message system, document processing, and database support, etc. They are mainly used for document-centric applications. However, these products has some serious limitations:

- No notion of *transaction* and locking or logging

- No higher level process definition capability

Groupware provides only *ad hoc* workflow capabilities. Many workflow models are designed to address these problems, based on the concepts of extended transaction models.

Extended transaction models such as nested transactions, multilevel transactions are too rigid for workflow applications. A basic problem with the development of workflow management systems based on a particular transaction model is that a predefined set of properties provided by the model may or may not required by the semantics of the workflow.

Another problem with adopting extended transaction models for designing and implementing workflows is that the systems involved in the processing of a workflow (processing entities) may not provide support for facilities implied by a transaction model.

To overcome these limitations of groupware and extended transaction models the workflow models are developed. The workflow system should support multi-task, multi-system activities where

- Different tasks may have different execution behavior and properties
- Tasks may be executed on different processing entities
- Application or user defined coordination of the execution of different tasks including data exchange is provided
- Application or user defined failure and execution atomicity are supported.

Workflow models are, in fact, seen as the future of advanced transaction models [2]. Some of the issues related to workflows are discussed in the paper on long running activities is found in [3, 4].

Some of workflow products developed in early systems are e-mail based. In these systems the jobs are routed through e-mail. But, the transactional properties of these systems are mostly lacking.

Exotica project at IBM's **Almaden Research Center** [9] is focused on the *scalability* and *availability* of the workflow systems. **FileNet** developing a commercial product [7] on workflow systems.

The **Workflow Management Coalition** [14] is developing a set of standards for workflow software to increase the interoperability and connectivity between different workflow products. Some of the current workflow systems are developed on the *World Wide Web* such as Action-tech [1] product. The advantage of web applications is, they allow tasks to spread wide across the Internet. A database located at any site can be used by other machine on the Internet.

1.3 Organization of the thesis

This thesis deals with the object management and tools of a workflow. The task management part of it is discussed in [13]. These two theses together cover the whole of design and implementation of a workflow system. These two thesis are closely related; so there are implicit references to [13] throughout this thesis.

The rest of the thesis is organized as follows :

Chapter 2 describes the basic concepts of a workflow model and a workflow management system. It presents a conceptual overview of a workflow system.

Chapter 3 presents the architectural overview of a workflow model by describing its components and their functionality. The proposed workflow model is described here in addition to the WfMC reference model.

The next chapter discusses in more detail, the design of the *object model*, which deals with the object definition language.

The object management, queue management and implementation of object server are described in detail in chapter 5 titled *The object server.*

The workflow specification and monitoring tools are described in chapter 6 titled *Workflow tools.*

We conclude the thesis with chapter 7 and discuss possible extensions to this work.

CHAPTER 2

Workflow Concepts

The main goal of this chapter is to provide a better perspective of the relationship between advanced transaction models and workflow models. First, we present some concepts of advanced transaction models and also the drawbacks of these models. Then we discuss the concepts of workflow model and application of this model through an example.

2.1 Advanced Transaction Models

Many of the early transactions (traditional transactions) are endowed with the atomicity, consistency, isolation and durability (ACID) properties. Many newer transaction models such as *Sagas* [8], *nested transaction model* [12] and *flexible transaction model* [5] are proposed, but only a few of these models have been implemented as prototypes and almost none are being used in a commercial product. These transaction models are too database-centric, which provides a nice theoretical framework but limits the possibilities and flexibilities of the models. Furthermore, since they tend to remain theoretical models, they generally ignore a large number of important design issues [2]. Many of the new applications are heterogeneous and distributed over a wide geographic area. They raise a variety of issues that are not addressed by transaction models.

The major difference between workflow and transaction models is in the area

of *correctness* and *reliability*. Current workflow systems do not have significant support for recovery and failure handling. In most cases user intervention is required. Transaction models, on the other hand, are in many cases motivated by these issues and many solutions have been proposed. However, it must be noted that most of these advanced transaction models are not implemented, the feasibility of these solutions are unclear.

2.2 Workflow Models

A *workflow model* is the representation of the type of the process, consisting of tasks (steps, activities) and their dependencies along with the data handled by the process. All these can be specified as a directed acyclic graph in which nodes represent steps of execution and edges represent flow of control and data among the different steps.

- *Workflow* is a collection of steps organized to accomplish a particular goal. It consists of tasks and relevant data and should have the start and termination conditions.

- *Task* defines a step within a workflow. A task is described by a set of states which mark logical steps in its execution. It passes through the states, depending on the conditions of execution. Each task has a name and a set of input and output parameters. Tasks are connected by two types of arcs : one is *data flow* arcs and another is *control flow* arcs.

- *Flow of Data* specified through data flow arcs between the tasks. These arcs maps the output parameter(s) of a task to input parameters of one or more tasks enabling them to exchange information.

- *Flow of Control* specified by control flow arcs between tasks, is the order in which tasks are executed. A control flow arc is associated with a condition. This is useful for conditional execution of steps.

- *Conditions* specify the circumstances under which certain events will happen. Conditions define labels of the control flow arcs and specify whether the control

flow should take place or not.

Workflow models have, in general, richer semantics and are more suitable to be used in commercial products. In many aspects workflow models are superset of advanced transaction models, this has been shown by implementing several advanced transaction models using a single workflow system in [2].

A *Workflow Management System* (WFMS) provides support for modeling, executing and monitoring the workflow. The important feature of WFMS is their ability to represent an organization and adapt the definition and execution of workflow processes to the particular characteristics of the organization. WFMS uses four different sets of entities : *users, tasks, programs, and data.* It controls and automates the interactions between element of each set. As outlined above, the WFMS automates the flow of control and data between the tasks, and maps activities to uses and programs. Current advanced transaction models solve only part of the problem. For instance, Sagas [8] provide a limited form of flow of control and data between tasks, but lack of reference to users or programs. The ConTract model [6] provides flow of control and data. but does not include users into the system.

In a WFMS, the organization is described in terms of hierarchical levels, roles and persons associated with it. A person can play several roles such as manager, programmer, etc., and a role can be assigned to several persons. The workflow designer must specify the users and roles responsible for the execution of tasks. If a role is specified, all the persons fit in that role are eligible to execute the task. This provides a greater flexibility in executing the workflow.

The tools are used to monitor the execution of tasks, validate individual tasks to detect problems such as redundancies, and tracking the location of documents. While monitoring one can stop a task, restart it, force it to finish, independent to the rest of the workflow. Each user interacts with the system using *worklists*. A worklist is a list of pending jobs of the user. A job can appear on several worklists simultaneously, if a user selects a job from the worklist it is automatically deleted from all other worklists. This can be used to perform load balancing in the execution of a workflow.

2.3 Example : A Bank Transaction Application

Consider a simplified example of cheque processing system in a bank. The workflow design for the cheque processing is represented graphical form in figure 1, which consists some of the following tasks:

- Scan the cheque or pay-in slip (or other forms)
- Extract the fields from the image
- Verification of the signature and account balance
- Update the account (retrieving or transfer of money), depending on the type of the transaction

In addition to the above tasks, we require some auxiliary tasks for processing. It is assumed that all the information about account holders are recorded in the database. This data is external to the workflow system.

The activity starts from the SCAN task, which controls the scanner. This task processes the documents and produces the image files. Related documents are entered into a *folder*; which is a container of related objects. This is used to maintain the relationship between the documents even though they are processed separately. For example, for transfer of money from one account to another, two documents are needed. One is the cheque corresponding to the source account and another is the pay-in slip for the target account.

The next step is the CLASSIFY task; which extracts the image file to identify various fields, using some image processing techniques. The type of the document (cheque, pay-in slip, etc.) is identified. A *work object* is created with the data associated with the fields. Depending on the type of the document, the work object is routed to different tasks. For simplicity, we discuss only the cheque processing. Some of the fields identified for the cheque are the account number, signature of the account holder, date on which cheque is drawn, value of the cheque and *bank code*. This code is used to identify whether cheque requires local or remote processing.

If the cheque requires an interaction with remote bank, it has to be sent there for processing. This may take a long time to complete. For local processing, we have

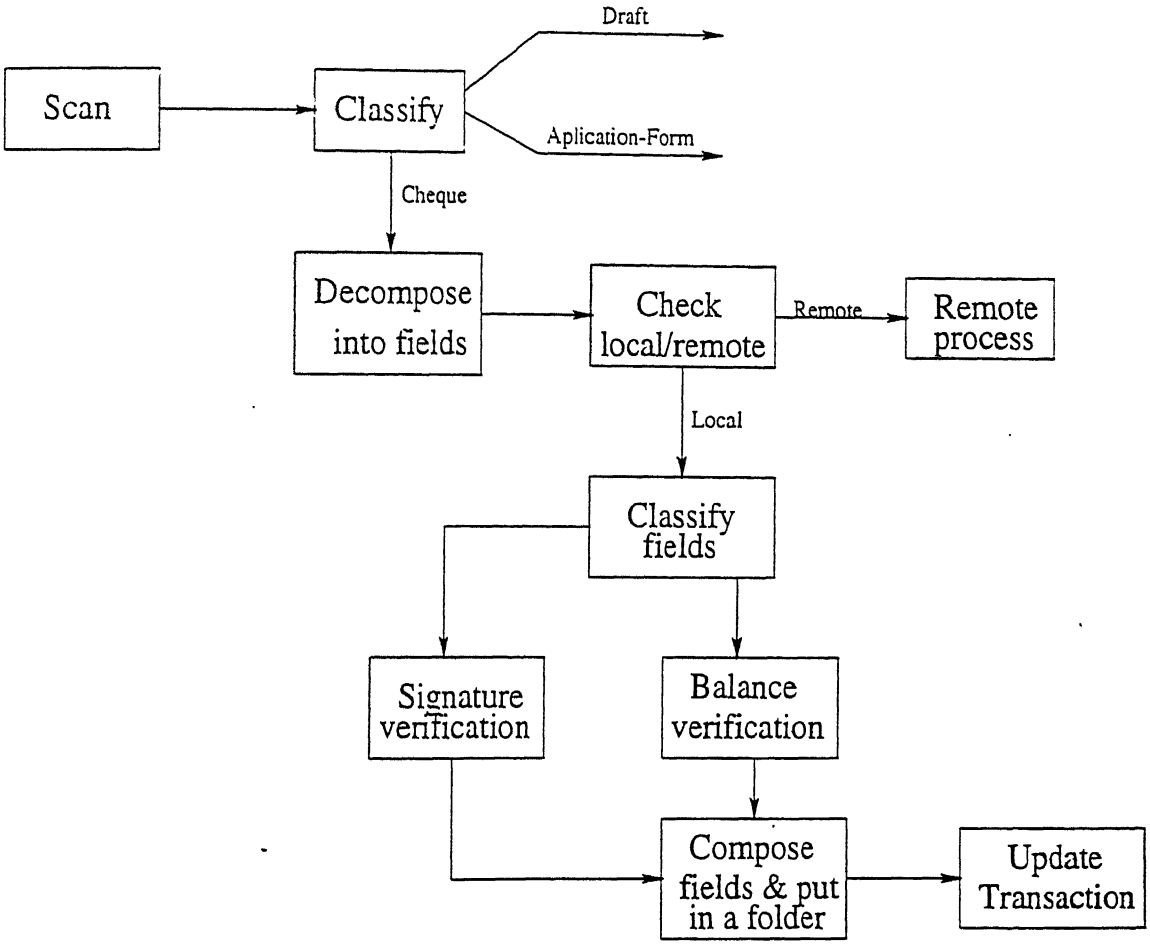


Figure 1: Workflow of a banking application

to verify the signature and balance in the account. These two tasks can be done in parallel. This is possible by creating sub-objects and routed independently to the different tasks.

After completion of the verification, the sub-objects are combined at the COMPOSE task. If the transaction is a single cheque processing, it forwards the object to UPDATE task. Otherwise, it waits for the other related documents before forwarding it. The updating done accordingly; and at this point the workflow is committed.

CHAPTER 3

System Architecture

This chapter gives the architectural overview of our workflow model. We first present the model for standard workflow system, later we describe the workflow model designed by us.

3.1 The *WfMC* Reference Model

The Workflow Management Coalition (**WfMC**) was founded in August 1993 "to promote the use of workflow through the establishment of standards for software terminology, interoperability and connectivity between workflow products" [14]. The standardization work is mainly the Workflow Reference Model shown in figure 2; which aims to "specify a framework for workflow systems, identifying their characteristics, functions and interfaces". The interfaces specified are :

- Process Definition Tools
- Workflow Enactment Services
- Workflow Client Applications
- Invoked Applications
- Administration and Monitoring

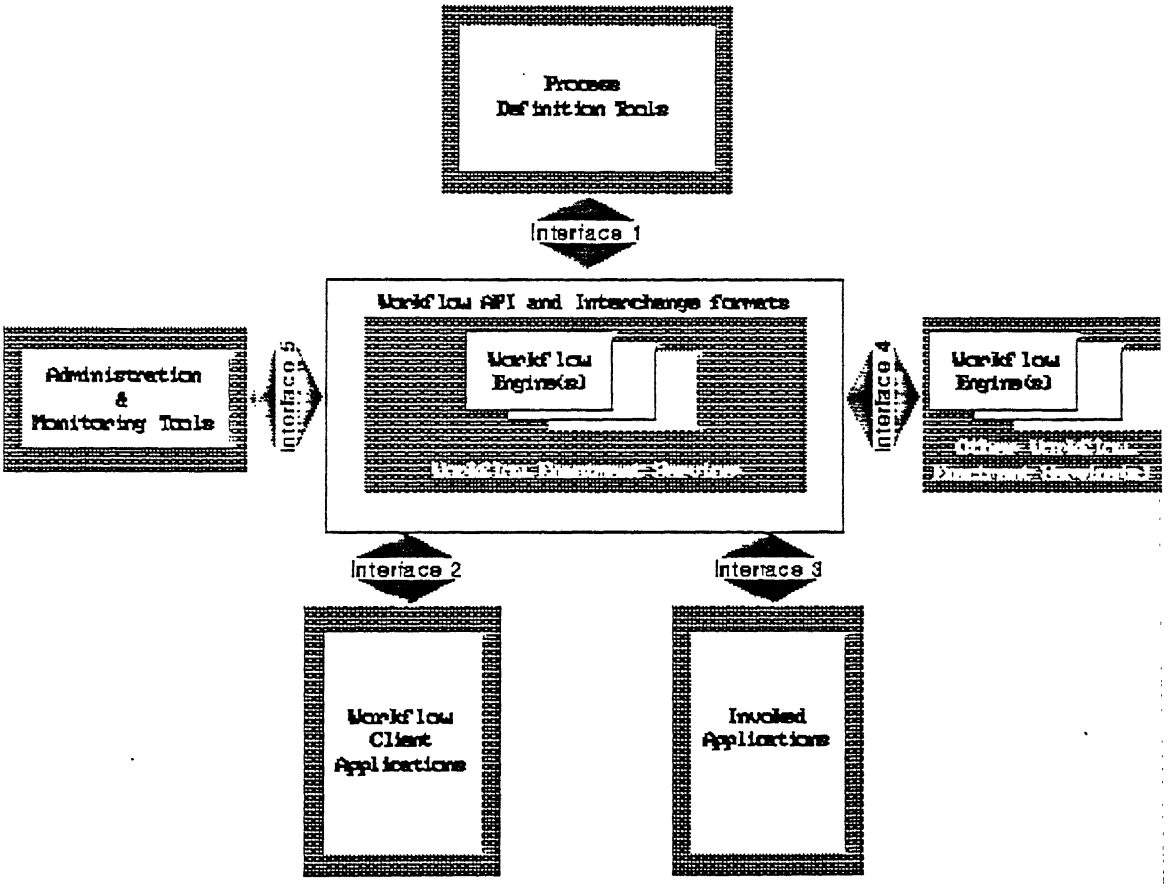


Figure 2: WfMC Reference Model

1. **Process Definition Tools**

   These tools are used for analyze, model, and describe the workflow. The interface provides common interchange format for importing/exporting the workflow definitions produced by the process definition tools. This makes the possibility of transferring the process definitions between different products and allow to define and execute processes using separate tools.

2. **Workflow Enactment Services**

   The workflow enactment service provides run-time server capable to create, manage and execute workflow process instances. The workflow enactment service may consists of one or more workflow engines providing the runtime environment. The primary concepts of WfMC is to provide interoperability between heterogeneous workflow enactment services allowing parts of the process execution to be passed to other workflow enactment services.

3. **Workflow Client Applications**

   The workflow client applications, allowing the users to select the tasks, retrieving details about the work to be done and invoke applications needed to perform the work among the other things.

4. **Invoked Applications**

   Wide range of pre-existing services in the heterogeneous environments must be integrated with the workflows. Invocation of automatic tasks should also be standardized.

5. **Administration and Monitoring Tools**

   The interface allows to administer all the workflow enactment services with the same tool. Similarly, monitoring and analyzing all the workflows would be enabled with one and the same product.

As these standards are still evolving, they are not discussed here in detail.

3.2 The Proposed Workflow Model

The schematic diagram of the proposed workflow model is given in figure 3.

A workflow designer starts with the *development toolkit* to describe the workflow. This includes identification of tasks, inter-task execution dependencies and data flow dependencies, as well as the termination conditions of the workflow. This is compared with the *process definition tools* mentioned in the standard reference model (section 3.1). This specification of the workflow is given to the task server.

The task server processes workflows according to the specification by submitting various tasks for execution, monitoring various events, and evaluating conditions related to inter-task dependencies. These tasks run as separate processes; and is equivalent to *workflow client applications* in the reference model. The workflow monitoring tool interacts with the task server to present continuous report of the execution of tasks and the flow of work objects. The user can see how a work object is getting routed and its status.

We designed a language for defining the work objects in the system. This language is called *object definition language* or *workflow language.* The **class** abstraction is used for defining the objects. This language is also used for writing the task code; for this **Task** construct and some other built-in functions are used. The work objects in the system are compiled into intermediate code called *workflow assembly language* (WASM), which can be executed on any machine.

Each task in the workflow has an interpreter for the WASM. This will be used to execute methods on the objects as they come to the task. A task can invoke an *external applications*, which may be full-fledged programs by themselves.

The object server, queue server together with the task server provides *workflow enactment service* mentioned in previous section. The object server provides storage and retrieval of objects. It uses RDBMS for its storage and this provides object-oriented interface to the clients. The object server also used as an interface for passing database objects, these are also called *associate objects.* These are external database objects and are different from the work objects.

The queue server maintains the queues of objects for all tasks. The queues are identified by the type of the task. This means that all task of the same type share

the queue. The queue server has lock management capabilities. It utilizes the object server for storing and retrieving the objects in the database.

Using the *workflow monitoring tool* user can visualize the flow of work objects in the system. This tool can also be used for querying the workflow in a variety of ways. For example, getting the current location of the work object, or finding the history of the task.

The communication between task server and object server is done by means of *remote procedure call* (RPC).

The next two chapters discuss the detailed design and implementation of the object language and the object server. Last chapter describes how tools are used for specification and monitoring the workflow. The task management and task server are discussed in [13]. For the sake of completeness, we present an overview of the task model in the next section.

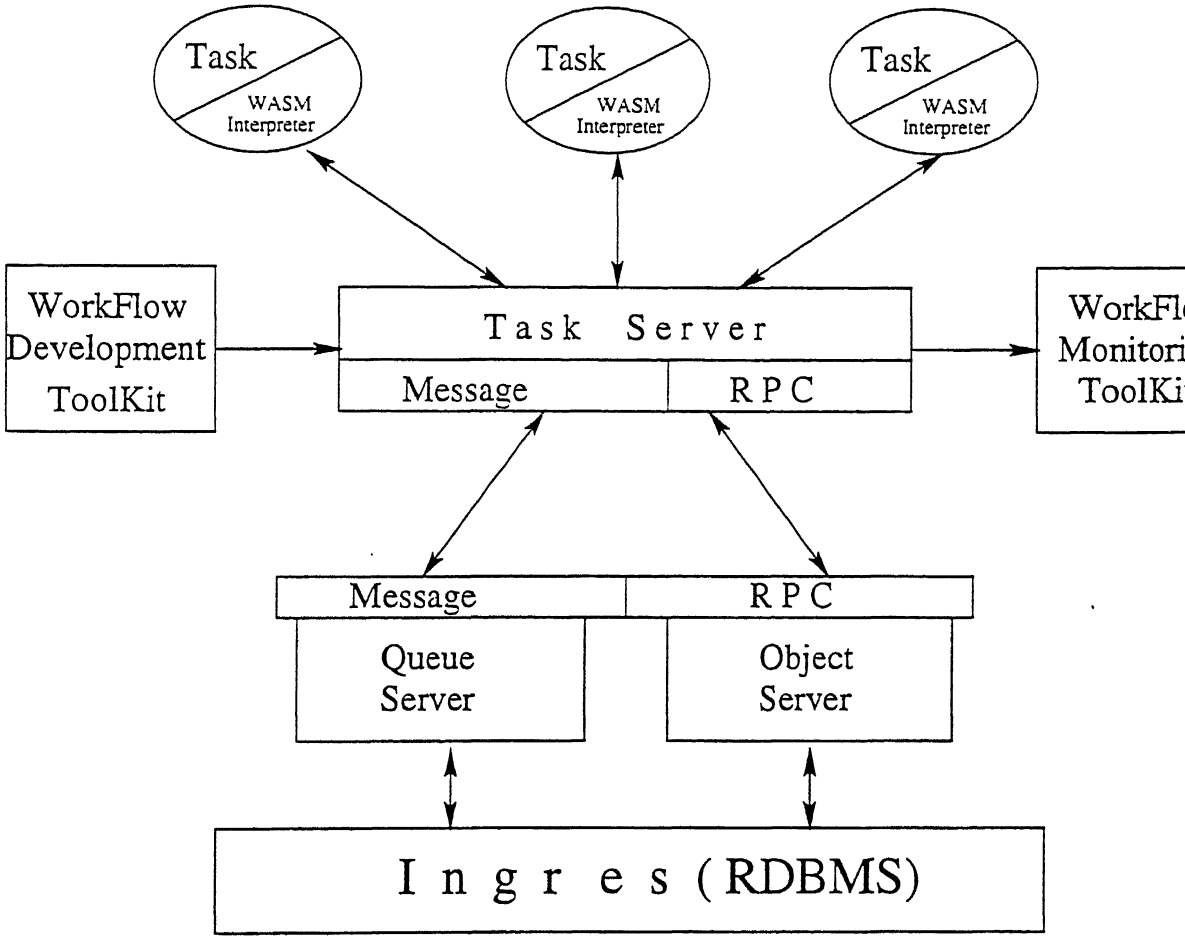


Figure 3: The Proposed Workflow Model

3.3 An overview of the Task Model

The definition of tasks, their controlled execution and scheduling are the main features covered in the task model.

A task is defined as a state transition system. As it processes the work objects, it passes through a sequence of states and terminates in one of the pre-defined final states. These states correspond to the possible terminations of a task are COMMIT of the task, COMMIT of the current workflow, ABORT of the current task and an exception.

Tasks interact among themselves by means of *events*. The list of users who can run a task, their roles, the method of invocation, the compensation specification and routing specification form a part of the *Task Specification*. It is written in a Task Specification Language.

The *Task Server* controls the execution of tasks by scheduling and delivering events to them. The task server monitors the state of execution of each task, by keeping a log of all activities in the system.

CHAPTER 4

The Object Model

The work objects in the system are defined in an *Object Definition Language* (or *workflow language*), a programming language with constructs for defining classes. The language is expressive enough for writing the task code. The main features of the language are the `class` construct and the built-in types `set` and `container`. The syntax and semantics of most constructs are same as that of `C` language; hence all the features are not discussed here in detail.

We need to compile this into intermediate code which can be executed on any machine. This is an essential requirement in order to migrate the work objects and execute methods on them at any task.

The language is used in a number of places in the system apart from the object definitions : It can be used to

1. write the ECA condition and action code segments
2. define the ASSERT and LEAVE code
3. specify the routing scripts

4.1 Language Features

4.1.1 The class construct

The class construct allows the programmer to define an encapsulation of attributes and methods; in other words, variables and procedures/functions. This is the basic mechanism for defining objects.

Here is an example for the definition of the class for Cheque in our bank application may look like this :

```
class Cheque {
        string  name;
        int     acctno;
        int     amount;
        int     bank_code;
        date    dd;             /* Date of issue   */
        Image   sign;           /* Signature image */

        bool isExpired() {
                if (dd + "00/06/00" < curdate())
                        return true;
                return false;
        }
        :
};
```

The isExpired() method determines the validity of the cheque (here, a cheque is valid for six months).

4.1.2 Task construct

This construct is used for defining tasks in the workflow. This has been described in section 4.4 of [14].

4.1.3 Other Programming Language Constructs

- conditional if statement

- `for` loop
- `while` loop
- `iterate` construct for iterating over the elements of a set or container.

4.1.4 Data Types

Several data types are provided as built-in types in the language such as `integer`, `float`, `char`, `boolean`, `string`, `date`. All the usual operations on variables of these types are allowed. In addition, there are two data types that are very important for workflow applications - **sets** and **containers**. The following two sections describe them in detail.

4.1.5 Container semantics

A **Container** is a finite ordered collection of homogeneous elements (i.e, data with the same type).

```
class Folder {
        Container<Form> list;
};
```

The type of a container is related to the type of the objects (containees) that are allowed to contain. Two containers are equal when they meet three conditions. They must have the same containee types and the same cardinality. Their elements must also be pairwise equal, in the correct order.

Every containee gets a unique sequence number in a container, which determines the order of arrangement of the containees. As a new containee is inserted before some existing containee, it changes the sequence number of the all subsequent containees. Similarly for the deletion operation also. Changing of the sequence number for simple operations is costly. Efficient implementation of the containers can be done by using a linked list. In a linked list, containees are not assigned any explicit sequence number. In the database, a *container table* is used to store the containers.

This table consists of the class name, container ID, and the sequence of containee object IDs.

The operations on containers are described below :

- *Adding an element to a container*
  There are three ways to add an element to a container.

  - `append(containeeID)`
  - `insertafter(position, containeeID)`
  - `insertbefore(position, containeeID)`

  The append function will add an element at the end of the container. Note that the sequence number of the new element is implicitly defined. Similarly, other two functions will also add an element but with a specified position, this is specified by a number. This functions will change only the logical sequence number of the subsequent containees, already existing elements are not affected.

  In the following example `list` is a container of type `Form` and form1, form2 are two types of Forms, `type` is an attribute in Form. form1 is appended to the container by append function, and form2 is inserted after the third containee in the list.

```
Container<Form> list;
Form    form1, form2;
        ⋮
form1.type = 3;
list.append(form1);
list.insertafter(3, form2);
        ⋮
```

- *Removing an element from the container*
  The function `delete(containeeID)` will remove a containee, given its ID. Only the logical sequence number of the subsequent containees are changed.

This will return an error if the containee not found in the container. For example, `list.delete(form1)` removes `form1` from the container `list`.

- *Detaching and attaching the containees*

  In general, a container contains several objects, which may be processed by several tasks independently. If a task wants to send a containee to another task, it detaches it from the container and sends to it. These detached objects can be attached after processing.

  The `detach(containeeID)` function detaches the containee by marking it as detached from the container. The detached object is still a part of the container but it cannot be accessed until it is attached again. The `attach(containeeID)` function is used to do this. For example, `list.detach(form1)` detaches `form1` from the container `list`.

- *Querying the container*

  `query(attribute_name, regularexpression)`

  The `query()` function is used to get the selected objects from the container, by matching the regular expression on some attribute of the containee objects. A regular expression over attribute values can be defined to qualify containees during retrieval. This function will returns an output container with the matched containees.

  In the following example `list` is container of type Form, and type is an integer attribute in Form. The query is to find all the forms whose type starts with 2 and ends with 3. The output of all the matched containees is returned into `listout`.

```
Container<Form> list, listout;
        ⋮
listout = list.query(type, "^2*3$");
```

The `query()` function is executed at the client, only if the full container is available there. Otherwise, it is passed to the server to be execute the query immediately (synchronous call).

- *Printing the container*

  The `print()` function will display the object attribute names and corresponding values of all containees in the container.

- *Finding the position of an object and vice versa*

  The function `getcontaineeid(object)` returns the sequence number of the specified object. It returns a negative number, if object is not found.

  `getcontainee(position)` is used to get the containee with specified sequence number.

- *Iteration over a container*

  We can traverse the whole container using the functions `getfirst()` and `getnext()`. This provides a way to define an operation on an aggregate type that can be called multiple times and that will return a different member each time. This is an essential feature for operating on a container, since a container may be huge and cannot always be kept in main memory.

- *Checking for detached objects*

  `iscomplete()` is used to check if there are any detached objects in the container. It returns `true` if all the containees are present locally and no object is detached. The following example shows the usage of some functions described here.

```
Container<Form> list;
Form    form1, form2;
int     position;

:
list.print(); /* prints the container objects */
position := list.getcontaineeid(form1);
form2 := list.getcontainee(position);
if ( list.iscomplete() ) {
     write("No Detached Objects in the container");
}
form1 := list.getnext();
:
```

4.1.6 Set semantics

A Set will mostly be used for representing two-dimensional tables in "forms". Each column is predefined scalar of various types, that is basically same as a tuple in a First Normal Form(1NF) relation in the relational data model. < "kala", 20> is an instance of type tuple <string, integer>. A type has a name and it contains a structure and a set of methods, applying to these objects. A structure will be a basic types such as string, integer, float etc. Each row of the set represents a tuple and each column represents values of a single type.

- *Adding an element to a set*
  `insert(<tuple>)` will add the tuple as a element to the set. In the following example two objects (tuples) are inserted into person set.

```
Set<string, integer, char> person;
        ⋮
person.insert("kala", 23, 'M');
person.insert("kommu", 20, 'M');
```

- *Removing an element from the set*
  The function to remove the tuple is `delete(<tuple>)`. This removes the object from the set, if the object is present, Otherwise it prints an error message.

- *Traversing over a set*
  We can traverse the whole set using the methods `getfirst()` and `getnext()`. This method will return one tuple at a time. An example for this is given below:

```
string  name;
char    sex;
int     age, flag;
        ⋮
flag = person.getnext(name, age, sex);
if (flag == 1) {
          write("Name = ", name);
```

```
            write("Age = ", age);
            write("Sex = ", sex);
    }
        :
```

- *Finding an element in the set*
  To test whether an tuple exists or not, use the method `find(<tuple>)`.

- *Printing the contents of the set*
  The method `print()` prints the contents of the set, that is the value of the each tuple is displayed.

4.2 Workflow Assembly Language (WASM)

The classes and methods are compiled to an intermediate code, called the WASM code. The tasks store the intermediate code of each method of a class, while storing the schema of the class. Whenever a task wants to execute a method it first gets the intermediate code using the call `schema_getmethod()`. It is then interpreted by the WASM interpreter. This code is for a virtual stack machine.

The WASM interpreter can call built-in functions or external functions linked to it. The interpreter can be extended any time with new functionality without touching the syntax of the intermediate code.

4.2.1 The stack machine

The high-level programming language statements are translated into instructions for an abstract stack machine. The architecture of the machine is briefly discussed below.

Logically, two stacks are used by the stack machine. One is the stack used on the traditional machines for typical programming language implementations. It stores the Activation Records (function arguments, return address, etc.). The other one is a small stack used by the processor itself as a substitute for registers. It is used for storing the operands of the most primitive instructions like add, sub, etc.

The instruction set for the machine is described below :

- `start` <*address*>
  Entry point for the program.

- `const` <*type*> <*constant*>
  Push a constant onto the stack. The first argument gives the type.

- `pop`
  Pop the topmost symbol off the stack. It is ignored.

- Arithmetic operations
  `add, sub, neg, mul, div, mod.`

- Logical operations
  `gr, ge, lt, le, eq, ne, not, and, or.`
  All these binary operators take their arguments from the stack. After the operation, the arguments are replaced by the result.

- `assign`
  Assign to a variable. The lvalue and the rvalue are the top two symbols on the stack.

- `if` *then_ptr [else_ptr] next_ptr cond_code*
  The conditional statement. The arguments are addresses of the appropriate instructions.

- `while` *body_ptr next_ptr cond_code*
  The while loop.

- `for` *index_code start_code end_code body_code next_ptr*
  The for loop.

- `alloc` <*function*> <*nargs*>
  Allocate space for <nargs> symbols on the stack. They all be used as the actual arguments for <function>.

- `call` <*function*> <*nargs*>
  Call a procedure or function. <function> is the function index. The schema is looked up to locate the function symbol and details. nargs is the number of arguments (which are already on the stack.

- `procret` and `funret`
  Return from a procedure or a function.

- `arg` <*n*>
  Push the *nth* argument of the current function onto the stack. This will be used as an operand in a subsequent operation.

- `bltin` <*function*> <*nargs*>
  Call the builtin function. <nargs> is the no. of arguments passed to it.

- `member` <*n*>
  Push the *nth* attribute of the current object ("This") onto the stack.

- `dot` <*n*>
  Get the *nth* attribute of an object (which will be on the stack). This is similar to 'member' except that it operates on any object.

- `stop`
  This is like a NOP. It is used to indicate the end of a loop body, etc.

CHAPTER 5

The Object Server

All the objects in the system are maintained by the *Object Server*. It provides a consistent interface to the tasks and the queue server for storing and retrieving objects. (This API is described in detail in 5.3). There are two kinds of objects the object server maintains :

- **Work objects** : These are the *primary* objects in the workflow that are passed through the tasks and serviced. As described earlier, a task will need to retrieve them and store them after processing. The tasks interact only with the queue server for work objects. And the queue server uses the object server for the purpose.

- **Associate Objects** : These represent the additional data that is needed by the tasks while processing the work objects. The database objects are usually represented in this manner. The task server exchanges these objects with the object server by means of the API.

  The storage and retrieval of associate objects is done through a separate interface to the object server. Queries on these objects with regular expression-based selection are supported. Alternatively, one can have a direct interaction at the SQL level with the RDBMS. Unlike the work objects, recovery of these objects is supported only to a limited extent. This is because the WFMS usually has little control over the external database.

The object server uses an RDBMS as a repository for these objects. It has to transform the object structure into relational structures and *vice versa* for doing this. All these details are hidden from its clients.

The most important part of a task is execution of a method on an object which has arrived from the *Object Server*. The object itself contains only the data. The method code has to be obtained from the schema separately. The method code thus obtained is cached, so that further requests need not contact the object server.

Each task in the workflow performs some operations or executes some methods on its work objects. To do this, it should have the schema of the class (which includes the method code) available. It gets this by sending a request to the object server. The schema thus retrieved is stored in the cache. So, when the same class definition is wanted later, it can be obtained from the cache, thereby avoiding another request to the object server.

If a task wants a work object, it contacts the Qserver, which in turn contacts the object server, to get the object.

Most of the schema of the classes and methods are stored initially when the workflow is started. Still, a task can store a class definition at run time if schema not already present.

A system can have multiple servers. Each server will be given a certain range of object IDs, which it uses to assign to newly created objects. The clients will select the server based on the OID.

The object server essentially maintains three databases

- *SchemaBase*:
  It contains the schema related information of the classes and methods.

- *MethodBase*:
  Intermediate code of methods is stored in this database, so that method compilation is not required at the client side. Each task in the workflow is capable of interpreting the code, using the WASM interpreter.

- *ObjectBase*:
  The data for the objects (work objects and others) are stored here. The work

objects are not directly stored by tasks. Instead they send the objects to the Qserver. The Qserver assigns a *newobjectID* to the work object and then stores in the object server. Only objectID is put in the Queue.

The first two are described in the next section and the ObjectBase is described in 5.2.

5.1 SchemaBase

Programs in the workflow language are compiled into a set of class definitions and the intermediate code for methods. The format of the schema in main memory is as follows :

```
schema(Class)
{
      NAttributes     NMethods
      For each Attribute {
            If (type(Attribute) = Class) {
                  /* Pointer to schema of the Nested Class */
                  schema(Attribute)
            } else {
                  AttrName offset type
            }
      }
      For each Method {
            MethodName ReturnType Nparams ParamTypes MethodCode
            For each Parameter {
                  ParameterName offset type
            }
      }
}
```

This structure has to be flattened for storing in the relational database. The class attributes and methods are stored in tables in the database, such that the whole class definitions can be re-constructed from the tables, when requested. The class name must be unique in the class definitions. We store the basic class information

such as name, number of attributes and number of methods in a `class table`. At the topmost level, we store the number of attributes and methods for each class as a tuple in a table.

class table structure

ClassName	Nattr	Nfun

For each attribute of the class we store the name and type in the `class_attr table` indexed by the class name. The serial number field in class_attr table is used to maintain the sequence among attributes. Simple attributes are easy to store. For attributes which are objects by themselves, we store the name of its class along with the name of the attribute. This acts as a pointer to the schema of that class.

class_attr table structure

Sno	ClassName	AttrName	AttrType

The attribute names and types can be retrieved by executing the following SQL query by the object server. A cursor is used to get successive values from the result.

```
select   attrname, attrtype
from     class_attr
where    class_attr. classname =  cname
orderby  sno
```

In the above query, `cname` is the class name specified by the task. Attribute information for all the classes is stored in a single table indexed by the class name, to reduce the number of tables in the database.

Similarly, the *signature* of each method is stored as a tuple in the `class_fun table`. This includes the method name, return type, number of parameters and intermediate code length.

class_fun table structure

Sno	ClassName	MethodName	ReturnType	Nparams	Codelength

These are the only details about a method, in addition to the code, that are required to execute a method dynamically. The method code is stored in the table fun_code. This is made a separate table because the length of the code is variable. The intermediate code of the method consists of several instructions; storing these instructions as tuples into the database increases the redundancy, because we have to store two other variables *classname* and *methodname* along with the instruction code. The best way is by storing group of instructions in one tuple. All the merged into a single string and it is divided into blocks of 512 bytes. Each such block is stored in one tuple.

fun_code table structure

Sno	ClassName	MethodName	Code

The method parameters are stored in the fun_param table. The parameter information is useful to do type checking when the method is called. Method information for all the methods is stored in a single table.

fun_param table structure

Sno	ClassName	MethodName	ParameterName	ParamType

5.2 ObjectBase

The object data model distinguishes between *literal* objects, such as integer, string, real etc., and *non-literal* objects such as persons and departments. Literal objects are directly representable, whereas non-literal objects are represented internally in the

database by surrogate identifiers. The object server provides for explicitly creating and deleting non-literal objects and for assigning values to their attributes.

Objects belonging to the same type share common properties (or attributes). For example, all the objects belonging to the person type have a name and age attributes. Storing the objects is much like the treatment of entities and their attributes in the E-R model or like one use of the relational model, where a row represents an object and each column represents an attribute.

An object may contain references to other objects, but they are not considered to be a part of the representation. They are distinct objects with their own identity that can be freely referenced from other objects.

Let us consider the problem of mapping objects to relational tables. Here the number of tables we need and their structure is not known statically. A relational table structure (*schema* in the RDBMS sense) is to be created for each distinct class to store the object data. The fields of the table directly correspond to the attributes of the class. For simple attributes the values are directly representable. For non-simple attributes the ID of the object is stored as the value of the attribute. Each row in the data table is an object, and each column is an attribute. The objectID is stored along with the attribute data, which is the primary key of the object data table.

Objects in containers are also stored in the object data table, this objects can be distinguished by the `containerID` field in the table. This value is -1 for simple objects. The additional information of the container is stored a `container` table is as shown below.

container table structure

ContainerID	ConName	ClassName	Numobj	Cursor

This table contains the additional information of the container such as number of objects, current position and container name. In the next section, we describe how these tables are manipulated by the functions for storing and retrieving the schema.

5.3 Object Server API

Object Server provides the following calls to its clients.

- `Schema_Store`
- `Schema_Retrieve`
- `Schema_GetMethod`
- `Object_Store`
- `Object_Retrieve`
- `Query_Container`

1. *Schema_Store()*

   This is the function for storing the schema for a class, given the class definition structure described in section 5.1. We can directly represent the simple attributes in the database, but for non-simple attributes we have to store parent classname along with the attribute name. The data table for a class is created with the name "_data" prepended by the *class name*.

   This will do the following steps.

   ```
   Checks for the duplicate class
   Insert <name, nattr, nfun> into the  class table
   Insert <classid, attr_name, type> into  class_attr table
   If (type(Attribute) = Class)
         Store name as  ParentClassName::AttributeName
   If (nattr not zero)
         Create a Data table of the class, using the attributes
   Insert the method related information in  class_fun table
   Store the intermediate code in the  fun_code table
   ```

2. *Schema_GetMethod()*

   This takes the name of a class and a method as arguments and returns the method parameters, return type, intermediate code etc. Storing and retrieving the method code is described in section 5.1.

3. *Schema_Retrieve()*

This function takes classname as an argument and returns the schema of the class. There are two ways to return the schema for compound class (which contains another class as an attribute). One is to send the schema for every attribute class, the second is to send the name of the attribute class along with the name of the attribute. We have implemented the second method. The disadvantage in the first case is *redundancy* (sending the schema for every attribute class which already exists in the client side). In the latter case, the task will send further requests for the schema of the attributes.

```
Get nattr and nfun from the  class table.
For each Attribute of the  class_attr table {
   If (type(Attribute) = Class) {
               /* Get the schema for the nested class */
               schema_retrieve(name(Attribute))
   }
}
Get the method related information such as names of
methods, return types, number of methods etc.
```

This function will not return the method code. Instead, it sets a flag in the returned structure to indicate that the method code is not retrieved. It is requested separately by the task.

5.4 Queue Management

The queues of objects for all tasks is maintained by the *Queue Server*. The queues are identified by the type of the task, rather than the task ID. This means that all tasks of the same type share the queue. It forms a new objectID for objects that are entered into the queue for the first time. The structure of objectID is described in section next section. Each task specifies the locks on objects while retrieving the objects from the Qserver. This lock management is described in section 5.4.2.

5.4.1 Structure of the Object ID

Object Identity is that property of an object which distinguishes each object from others. Whenever a task creates an object, it will give an objectID to it, this we call it the *Original ObjectID*. These objects are put in the queues for further processing. In order to distinguish the object uniquely from other tasks and in the database, the Qserver creates a new objectID for it. This objectID includes a *serverID*, which is the identifier given to the object server, as there may exist more than one object servers in the workflow. Each object server has an identifier in the workflow, this we call it as serverID.

Each object will be given a unique number to distinguish it within the object server, because two different tasks may send objects with same original objectID. These numbers are maintained as a linked list. Whenever an object is deleted from the server, the number given to that object is added to the list.

ServerID	Server rangevalue	Original ObjectID

5.4.2 Lock Management

The Qserver supports locking of objects. By locking an object, a task can prevent the other tasks from accessing the object. When a task requests a work object from the queue, it specifies a lock on the object. The Qserver searches the objects within the queue and for each object checks the accessibility of the task to the object. If an object is found it gets the object from the queue; locks it (in the mode specified by the task) and returns it to the task. The following are the modes in which locking may be done :

Read-Only (LOCK_RO)
Append-Only (LOCK_AO)
Write-Only (LOCK_WO)
Read-Write (LOCK_RW)
Read-Append (LOCK_RA)

5.4.3 Table Structure of Qserver

The information for every object received by the Queue server is stored in *qserver table* in the database. The information includes the objectID, lock mode, queue type in which it is inserted etc., the actual content of the object is stored in the database by the object server. The queues are maintained by the Qserver using the *tasktype* as a field for every object in the qserver table.

The class name and the objectID are used to get the object data from the object server. The class name is required to get the name and type of each attribute in the class. Using the above information we can retrieve the object by using objectID as an index into the object data table.

Qserver table consists of `serial number`, `objectID`, `lockmode`, `task type`, `containerID`, `object name`, `class name` as fields.

For container objects the containerID is used to distinguish from other simple objects. A task requests the object by giving the queue type, the queue server searches for an object in the specified queue by executing the following SQL query.

```
select   objectid
from     qservertable
where   (qservertable. tasktype =  queuetype)
```

After getting an objectID the Qserver checks the locks on the object and retrieves the appropriate object.

5.4.4 Implementation of Server API

The Qserver provides API calls for storing and retrieving the objects in the database, and also for getting the selected objects from the database. The calls provided by the Qserver are:

- Object_Store
- Object_Retrieve
- Get_Container

Tasks use the function `Object_Retrieve()` to get the work objects from the Qserver. They perform some operations on the object and pass it to another queue, using the call `Object_Store()`. While storing, the task should specify the output queue name, in which the object is stored. The `Get_Container()` call is used to get a selected sequence of objects with certain property. Tasks may ask for a regular expression match on some attribute of the containees. This type of selection is useful for the following type of queries. Select the all bank holders whose account numbers starts with 2.

1. `Object_Store(Object, TaskType)`
   This function will store the object in the specified TaskType queue by the Qserver. Storing simple objects is done by calling the function `Store_Simple` directly. The information about containers (such as type, number of objects etc..) is first entered into the *container* table to get the containerID. The objects within the container and simple objects of same class are stored in the same table. Simple objects are distinguished by a containerID of -1..

   The steps followed by this function are:

```
Get the original objectID and construct new objectID as
discussed in section 5.4.1
If (Container)
     Make an entry in the  container table
Insert the tuple into the Qserver table
 <sno, oid, lock-mode, tasktype, conid, objectname, classname>
If (object is simple)
     call  Store_Simple(Object, OID)
If (object is container)
     invoke  Store_Simple() function for every object in the
     container
```

   Objects within the container are found by using `getnext()` method on the container.

2. `Store_Simple(Object, OID)`
   While storing objects this function traverses the attribute list of the class

object and it recursively calls itself for any non-simple attribute. Therefore this function can be used for storing nested objects.

```
Get the Attribute names, types from the  class_attr table
and insert the data into the corresponding data table as
<sno, conid, OID, ...data....>.
 If  (type(Attribute) is simple) {
       store the simple values in the data table
}  else {   /* Nested Object */
        store data as <classname::oid>
        call  Store_Schema(Object, OID)
}
```

The object data table for a class is created by Schema_Store() function. For example, the object table created for class Cheque described in section 4.1.1 is cheque_data. All the data associated with the class cheque is stored in the corresponding object table. The schema of the table is

```
< objectid:int,  conid:int,  name:string,  accno:int,
  amount:int,  bank_code:int,  dd:date,  sign:string>
```

Note that the type of the *sign* field is a string, here we store the objectid of *sign* along with the class name associated with it. This is used to reduce another query to the class_attr table to get the class name of the attribute *sign*.

3. Object_Retrieve(TaskType, Mode)

   The object retrieval proceeds as follows :

```
Get the first object in  TaskType queue from the qserver
table. (*Checking the lock-mode of the objects is required*)
Set the lock-mode of the object in the qserver table
Get the objectID and class name of the object from the table
Call the function  Get_Object(objectID, classname)
If (type(object) is container) {
      Retrieve the data for each object in the container,
      using  Get_Object(objectID, classname) function call
}
```

For containers, this function gets the containerID (say *foundconid*) from the *container* table, and performs a query on the object data table as follows:

```
select   objectid
from     cheque_data
where    cheque_data.conid = foundconid
```

A *cursor* is used for the above query to get all the objectIDs, and for each objectID we perform the above steps.

4. `Get_Object(OID, ClassName)`
   This function returns the object. While traversing the attribute list, if any non-simple attribute is found, it recursively calls itself using the objectID and classname of the attribute.

```
Get the attributes from the  class_attr table
 For each Attribute of the Class {
       If (type(Attribute) is simple) {
             Get the data of the simple attribute
       }  else {
             Get the classname, objectID  from the object table
             Call Get_Object(OID, ClassName)
       }
}
```

5. `Get_Container(ClassName, AttrName, Pattern)`
   This function returns the sequence of matched objects to the client in an ensemble structure. The *pattern* is a regular expression on some attribute of the class. The function does the following steps:

```
Get the  nattr and  nfun from the  class table
Create a temporary table to store the matched objects
For each object in the Container {
   compare the pattern with the data of the specified attribute
   If (match found) {
         Put the object into the table
   }
}
```

Only some of the matched objects will be sent at one instant. The object server creates a temporary table for storing the result of the query. It maintains a cursor on this table to know how many objects are already sent to the task. At the client side, these objects are accessed sequentially; a request is sent to get the next sequence of the objects after processing. This will be continued for all objects in the table, at which point the object server deletes its temporary table.

Before performing a search over the container in the database the object server will check for detached objects, by using the function call `IsContainerFull()`. It waits for the detached objects. The search is performed only when the container is full.

CHAPTER 6

Workflow Tools

This chapter describes two utilities which make the development and management of workflow easier. As mentioned in the workflow model of figure 3, these are

- Workflow specification tool
- Workflow monitoring tool

6.1 Workflow specification tool

The most significant part of the workflow is specification toolkit. Issues in specifying the workflow are:

1. Task specification
2. States of the task
3. Inter-task description

The workflow can be specified by using the *Task Specification Language.* Another way for specifying the workflow is by using a GUI tool, which will in turn generate the task specification language. Figure 4 shows a snapshot of the specification tool. The main window consists of two frames. The first frame consists of `jobname` (name of the current job) and `inter-task description` buttons. The second frame consists of `file`, `save`, `clear`, `task name` and `task description` options.

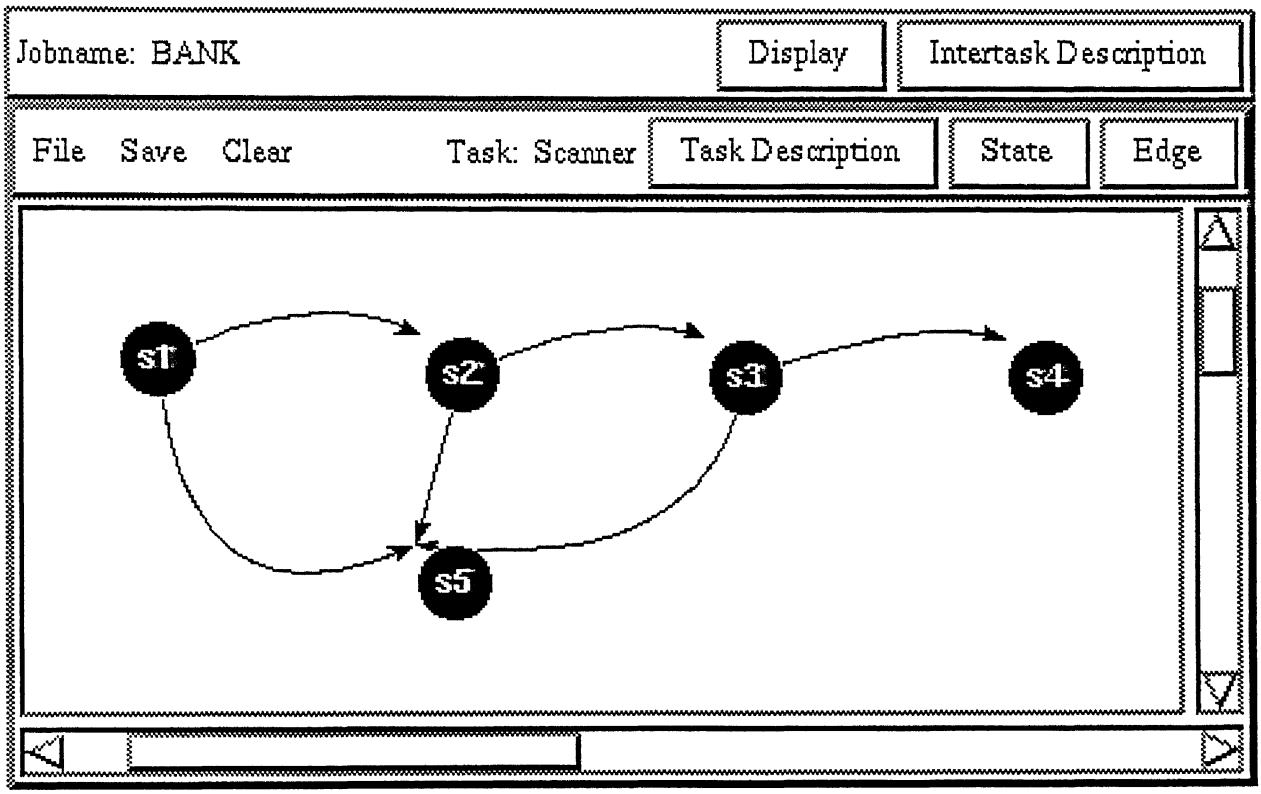


Figure 4: Workflow Specification Tool

- **Task Specification**

  An abstract model of a task is a *state machine* whose behavior can be defined by providing its *state transition diagram.* In general, each task can have different internal structure resulting in a different state transition diagram.

  For each task the following specifications are necessary, as mentioned in section 4.2 of [14].

  - TaskType
  - Users
  - Roles
  - Method of invocation
  - Compensated By
  - Compensation To
  - Task Code file
  - Routing Description

    The routing specification for the three pre-defined states is entered by means of a dialog box. It displays the following fields :

    * On COMMIT
    * On ABORT
    * FORWARD

    The `DISPLAY` button can be used to display the whole workflow. The tasks are displayed as rectangles and the edges as line between them.

  In our tool, a task is specified using the `TaskDescription` button. A dialog box is displayed with above options. Current task-type is displayed as `taskname` in the tool.

- **States of the task**

  A state can be *internal* or *exposed*; the task server only knows the exposed states. Transitions to internal states are not visible to task server. Inter-task

interaction is possible only through the exposed states. For each state of the task the following specifications are necessary.

- State Name
- Internal/Exposed
- Assert Code
- Leave Code

The state description is entered using the `state` button in the second frame of the tool and the edges between the states are drawn using `edge` button. Each state of a task is represented as a *circle* and the stateID is displayed on each state, so that we can distinguish one from another. The edges between the states are represented by *curves*. On clicking on the state, a dialog box containing the above options is displayed. The edges are specified by drawing a bezier curve between the two end points and intermediate points marked by the user. An example state diagram is shown in figure 4.

Once the task descriptions are over, we can specify the inter-task description as discussed in the next section.

- **Inter-task description**

  The inter-task structure of the workflow can be defined by specifying the inter-task description. Here, the dependencies between various tasks are specified.

  The event description is specified for two tasks at a time; these two tasks are displayed in a window, with their exposed states highlighted. Events between tasks are specified using inter-task edge. On clicking on the inter-task edge the above options are displayed. These edges are distinguished from the edges between the states, by using a different colour and width.

  The parameters to be specified for each inter-task edge are the following:

  - EventName
  - EventParameters (seperated by commas)

- Condition
- Action
- E-C Time
- C-A Time
- Mode(SYN/ASYN) : Mode shows whether event is synchronous, or asynchronous (eg. SYN).

6.2 Workflow monitoring tool

This tool is used to execute a query on the scheduler, to monitor the work objects in the workflow, through the User Interface. The tool window consists of three entries state, workobjectID and task (shown in the figure 5).

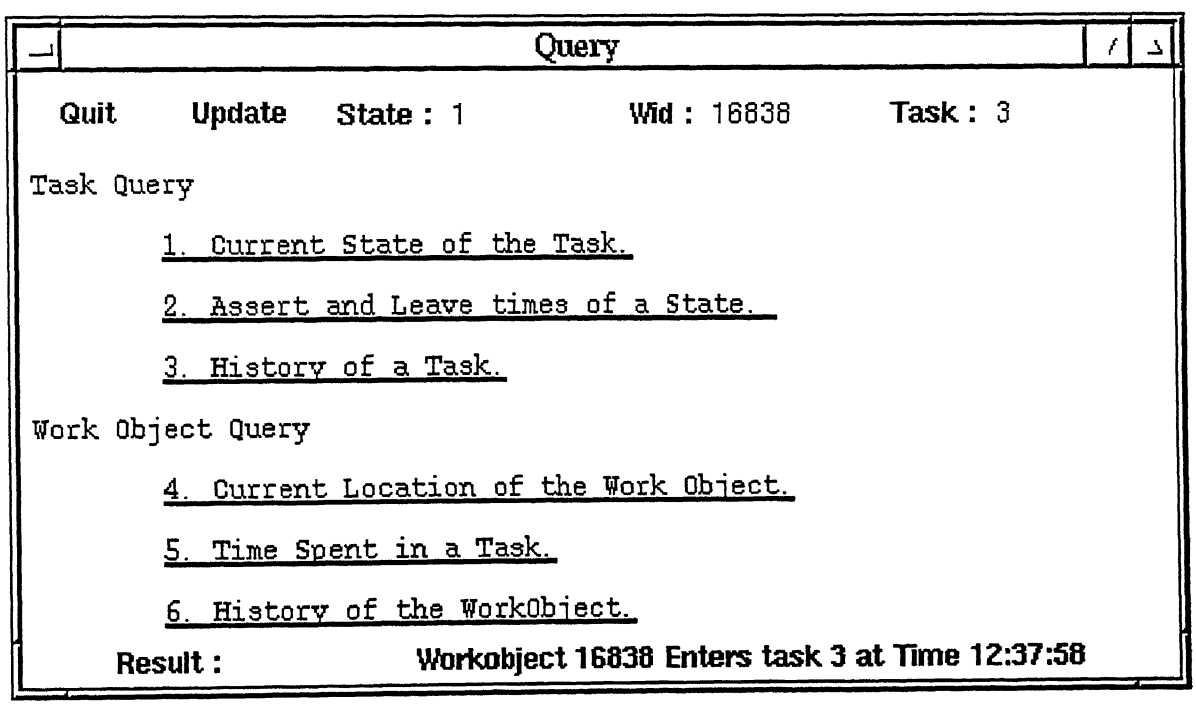


Figure 5: Workflow monitor/query Tool

6.2.1 Task monitoring

For each task the following options are available, these options are available in the tool as hypertext.

- *Current state of the task*

  This will display the current state along with the work object of the specified Task.

- *Assert and leave times of a state*

- *History of a task*

  This will display of summary of activities of the task since its invocation. In particular it displays the state the task asserts and the work objects it has processed (or processing) in the following format. All this information is displayed in a separate window.

  Task *ID* (type *type*) entered state *name* with work object *ID* at *date, time*

  Task *ID* (type *type*) raised event *name* to task *ID*

  Event *name* condition for task *ID* result : *result*

  Event *name* action for task *ID* result : *result*

6.2.2 Work object monitoring

The options available for querying the work objects are:

- *Current Location of the work object*

  The ID of the task and the state in which it is processing the work object.

- *Time Spent in a Task*

  This gives only the entering time of the work object into the task. This time will be displayed in the bottom of the window.

- *History of the work object*

  This shows the tasks in which the work object entered and the corresponding entering times. This will be displayed in another window in the following format.

  `Work Object` *ID* `entered task` *ID* `(type` *type*`) at` *date, time*

CHAPTER 7

Conclusions

In this thesis, we discussed the design and implementation of a workflow model. The workflow models are developed to overcome the limitations of groupware and advanced transaction models. In fact, workflow models can be seen as a future of extended transaction models. The object model and the object server parts of the workflow are considered in detail. Tools for specification and monitoring workflow are presented. The rest of the work is reported in the companion thesis [13]. An example application of workflows is described throughout the thesis.

We have implemented a prototype of the workflow model presented in this thesis. The implementation has been carried out with the help of a number of programming languages and tools. The major portion of the code has been written in C++. The UNIX tools *lex* and *yacc* are used for parsing the specification files. A full-fledged compiler has been developed for the workflow language. **Tcl/Tk** has been used for the graphical user interface programming. The UNIX mechanisms of RPC and message passing through sockets are used for communication between processes on different machines. **Ingres/Esqlc** embedded SQL C is used at the object server.

7.1 Future Work

Some of the extensions to this model and the implementation are discussed below :

- **Queue Model**

  The Queue Model can be made more sophisticated. At present, we have very simple queues which just store objects and retrieve them in a *first in first out* order. One possible improvement is the addition of *filters* at the ends of a queue. Only objects satisfying certain criteria are allowed to enter the queue. This way, we can enforce constraints on the objects that a task takes as input and produces as output.

- **User Interaction**

  In our system, only the workflow specification and the task specification can be generated visually. Graphical user interfaces can be used in the specification of objects also. It should be possible for the user to define the object attributes by selecting the data types from menus. Ideally, code for the methods of objects and tasks could also be generated visually. The generation of programs visually (or *visual programming*) is a separate research problem by itself.

- **Improvement in GUI**

  The Graphical User Interface can be significantly improved to get better user interaction and performance.

  A number of research perspectives in workflow systems with specific reference to the **Exotica** product have been mentioned in [11].

Appendix A

Workflow Language Syntax

```
1    %union {
2            Symbol  *sym;   /* symbol table pointer */
3            Inst    *inst;  /* machine instruction */
4            int     integer;
5            char    cval;
6            double  real;   /* Constants    */
7            char    *str;   /* Identifier   */
8    }
9
10   %token  BEG END VAR VIEW OF TO DO ASSIGN SET CONTAINER OF
11   %token  <sym>   WRITE WHILE FOR IF THEN ELSE BREAK
12   %token  <sym>   CLASS INTEGER CHAR REAL DATE STRING SET
13   %token  <sym>    RETURN FUNC PROC READ NEW
14   %token  ON ROUTE
15   %token  COMMIT ABORT FORWARD
16
17   %token  <str>   ID CONST_STRING
18   %token  <integer>       ARG CONST_INT
19   %token  <real>  CONST_REAL
20   %token  <cval>  CONST_CHAR
21
22   %type   <inst>  expr logical_expr stmt stmtlist
23   %type   <inst>  cond while for if end
24   %type   <integer>       exprlist param_list var_decl_list var_decl
25   %type   <integer>       member member_list id_list fun_defn_list
26   %type   <integer>       proc_fun type bltin_type user_type set_type
27   %type   <integer>       condition
```

```
28  %type   <str>           route_stmt
29
30  /* Operators in their increasing order of precedence */
31
32  %right  ASSIGN
33  %left   OR
34  %left   AND
35  %nonassoc '>' '<' GE LE EQ NE
36  %left   '+' '-'
37  %left   '*' '/' DIV MOD
38  %left   NOT
39  %left   UNARYMINUS
40  %right  '^'
41  %nonassoc FUNCALL
42  %left   '.'
43
44  %%
45
46  spec            :       object_def_list '.'
47                  ;
48
49  object_def_list :       object_def ';' object_def_list
50                  |       object_def
51                  ;
52
53  object_def      :       class_task__def
54                  |       error
55                  ;
56
57  class_task__def :       CLASS ID
58                          BEG var_decl_list
59                          fun_defn_list
60                          END
61                  ;
62
63  container_type  :       CONTAINER '<' ID '>' id_list
64                  ;
65
66  set_type        :       SET '<' type_list '>' id_list
67                  ;
68
```

```
69   type_list        :       type
70                    |       type_list ',' type
71                    ;
72
73
74   param_list       :       type ID
75                    |       param_list ',' type ID
76                    ;
77
78   var_decl_list    :       var_decl
79                    |       var_decl_list var_decl
80                    ;
81
82   var_decl         :       type id_list ';'
83                    ;
84
85   type             :       bltin_type
86                    |       user_type
87                    ;
88
89   bltin_type       :       INTEGER
90                    |       REAL
91                    |       CHAR
92                    |       DATE
93                    |       STRING
94                    |       set_type
95                    |       container_type
96                    ;
97
98   user_type        :       ID
99                    ;
100
101  id_list          :       ID
102                   |       id_list ',' ID
103                   ;
104
105  fun_defn_list    :       fun_defn
106                   |       fun_defn_list fun_defn
107                   |
108                   ;
109
```

```
110  fun_defn        :       type ID
111                          '(' param_list ')'
112                          BEG
113                                  stmtlist
114                          END
115                  ;
116
117  stmt:
118          RETURN
119          | RETURN expr
120          | BREAK
121          | expr
122          | while cond DO stmt end
123          | for expr
124            ASSIGN expr
125            TO expr
126            DO stmt end
127          | if '(' cond ')' THEN stmt end
128          | if '(' cond ')' THEN stmt end ELSE stmt end
129          | BEG stmtlist END
130          |
131          ;
132
133  cond:       logical_expr
134          ;
135
136  while:    WHILE
137          ;
138
139  for:      FOR
140          ;
141
142  if:       IF
143          ;
144
145  end:
146          ;
147
148  stmtlist: stmt
149          | stmt ';' stmtlist
150          ;
```

```
151
152  expr:    CONST_INT
153           | CONST_REAL
154           | CONST_CHAR
155           | CONST_STRING
156           | ID
157           | expr '.' ID
158           | expr funcall
159           | expr
160             ASSIGN expr
161           | '(' expr ')'
162           | expr '+' expr
163           | expr '-' expr
164           | expr '*' expr
165           | expr '/' expr
166           | expr '%' expr
167           | '-' expr   %prec UNARYMINUS
168           ;
169  logical_expr :
170             expr '>' expr
171           | expr GE expr
172           | expr '<' expr
173           | expr LE expr
174           | expr EQ expr
175           | expr NE expr
176           | logical_expr AND logical_expr
177           | logical_expr OR logical_expr
178           | NOT logical_expr
179           | '(' logical_expr ')'
180           ;
181
182  funcall:  '(' exprlist ')'
183           ;
184
185  exprlist:
186           | expr
187           | exprlist ',' expr
188           ;
189
190  workflow_spec:
191                    | routing_spec workflow_spec
```

```
192                     ;
193
194   routing_spec      : ON condition
195                       route_stmt
196                     ;
197
198   condition         : COMMIT
199                     | ABORT
200                     | FORWARD
201                     ;
202
203   route_stmt        : ROUTE ID ';'
204                     ;
205
```

Bibliography

[1] ACTION TECHNOLOGIES. http://www.actiontech.com/.

[2] ALONSO, G., AGARWAL, D., EL ABBADI, A., KAMATH, M., GUNTHOR, R., AND MOHAN, C. Advanced transaction models in workflow contexts. Tech. rep., IBM Almaden Research Center.

[3] DAYAL, U., HSU, M., AND LADIN, R. Organizing long-running activities with triggers and transaction. In *ACM SIGMOD Conference on Management of Data* (1990).

[4] DAYAL, U., HSU, M., AND LADIN, R. A transaction model for long-running activities. In *Proceedings of the 17th International Conference on Very Large Databases* (1991).

[5] ELMAGARMID, A., LEU, Y., LITWIN, W., AND RUSINKIEWICZ, M. A multidatabase transaction model for interbase. In *Proceedings of the 16th International Conference on Very Large Databases* (August 1990).

[6] ELMAGARMID, A. K., Ed. *Database Transaction Models for Advanced Applications*. Morgan-Kaufmann, 1992.

[7] FILENET TECHNOLOGIES. http://www.filenet.com/.

[8] GARCIA-MOLINA, H., AND SALEM, K. Sagas. In *ACM SIGMOD Conference on Management of Data* (May May, 1987), pp. 249–259.

[9] IBM ALMADEN RESEARCH CENTER. http://www.almaden.ibm.com/almaden/.

[10] KIM, W., Ed. *Modern Database Systems.* Addison Wesley Publishing Co., Reading, Mass., 1995.

[11] MOHAN, C., ALONSO, G., GUNTHOR, R., KAMATH, M., AND REINWALD, B. An overview of the exotica research project on workflow management systems. Tech. rep., IBM Almaden Research Center.

[12] MOSS, J. Nested transactions : An approach to reliable distributed computing. Tech. Rep. MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[13] SRINIVAS, K. Design and implementation of a workflow model, part I : Task model. Master's thesis, Dept. of CSE, Indian Institute of Technology, Kanpur, February 1997.

[14] THE WORKFLOW MANAGEMENT COALITION. http://www.aiai.ed.ac.uk/wfmc/.